escuelaingenieríaindustrial
PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO

Estructuras de datos en memoria principal

Franco Guidi Polanco
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso, Chile
fguidi@ucv.cl

Actualización: 11 de abril de 2006

Estructuras de datos

- Estructuras básicas
  - Arreglo
  - Lista enlazada
    - Simplemente enlazada
    - Doblemente enlazada
- Colecciones implementadas sobre las estructuras básicas:
  - Lista, Lista con iterador
  - Lista circular
  - Pila
  - Cola
  - Hashtable
  - Vector (Java)
  - (Otras)

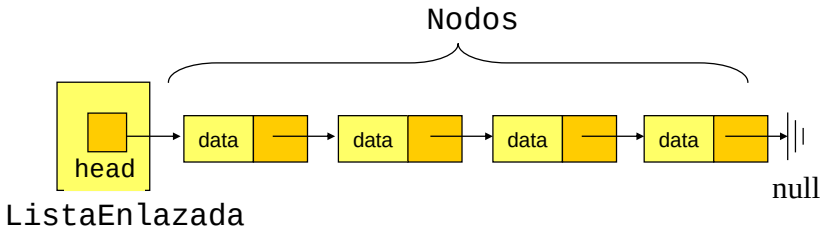
Arreglo

- Es una colección ordenada de elementos del mismo tipo.
- Es de largo fijo, definido al momento de instanciarlo.
- El acceso a los elementos se hace a través de un subíndice.
- Fácil de recorrer en ambos sentidos.
- Estudiado en cursos anteriores

Listas enlazadas

- Son estructuras dinámicas: se asigna memoria para los elementos de la lista en la medida que es necesario.
- Cada elemento se almacena en una variable dinámica denominada **nodo**.
- En la lista **simplemente enlazada**, cada nodo apunta al nodo que contiene el elemento siguiente

Esquema tentativo de una lista simplemente enlazada

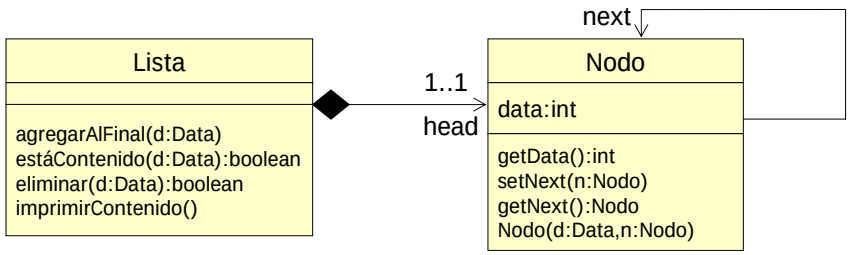


Datos contenidos en la lista

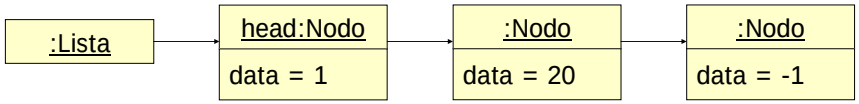
- Los nodos de una lista contendrán datos del tipo declarado en la estructura del nodo. Por ejemplo:
  - Tipos primitivos (byte, int, boolean, char, etc.)
  - Referencias a objetos
- En los siguientes ejemplos consideraremos el uso de listas de enteros, aunque las técnicas que serán descritas son aplicables a cualquier otro “tipo” de lista.

Diagrama de clases de una lista simplemente enlazada

- Diagrama de clases (lista de enteros):



- Diagrama de objetos:



Una lista simplemente enlazada (versión preliminar)

- Declaración de la Lista:

```
public class Lista{
   Nodo head = null;
   public void agregarAlFinal(int data){
      ...
   }
   public void imprimirContenido(){
   ...
   }
   public boolean estaContenido(int data){
   ...
   }
   public boolean eliminar(int data){
   ...
   }
}
```

Nodos en una lista simplemente enlazada

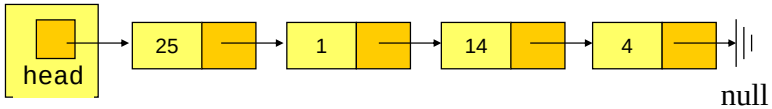
```
public class Nodo{
      private int data;
      private Nodo next;
      public Nodo(int d, Nodo n){
            data = d;
            next = n;
      }
      public int getData(){
            return data;
      }
      public Nodo getNext(){
            return next;
      }
      public void setNext(Nodo n){
            next = n;
      }
}
```


Inserción en la lista simplemente enlazada

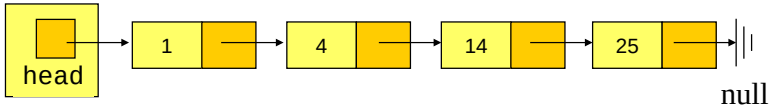
- Insertar elementos:

Ejemplo: [25, 1, 14, 4]

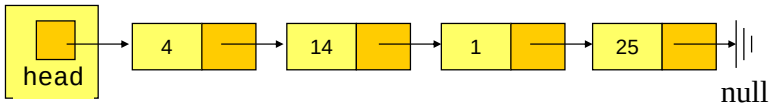
  - Al final de la lista



  - Manteniendo un orden

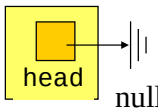


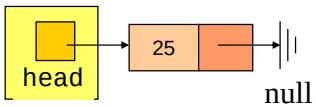
  - Al inicio de la lista



Inserción de elementos al final de la lista

- Caso 1: la lista está vacía (variable **head** contiene **`null`**)

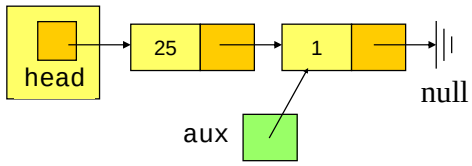




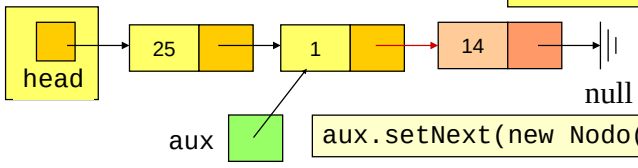

```
public Lista{
      ...
      public void agregarAlFinal(int dato){
            ...
            head = new Nodo( dato, null );
            ...
      }
      ...
}
```


Inserción de elementos al final de la lista

- Caso 2 (general): la lista tiene al menos un elemento



1. ubicar último nodo de la lista (aquél cuya variable next contiene null)
2. Instanciar el nuevo nodo con el contenido indicado
3. Asignar el nuevo nodo a la variable next del último nodo (asegurándose de que la variable next del nuevo nodo sea igual a null)




```
aux.setNext(new Nodo(dato, null));
```

Inserción de elementos al final de la lista

- Caso general:

```
public class Lista{
   ...
   public void agregarAlFinal(int dato){
      Nodo nuevo = new Nodo(dato, null);
      if( head == null )
            head = nuevo;
      else{
            Nodo aux = head;
            while( aux.getNext() != null)
                  aux = aux.getNext();
            aux.setNext( nuevo );
      }
    }
   ...
}
```


Recorrido de la lista

- Método que imprime el contenido de la lista:

```
public class Lista{
  ...
public void imprimirContenido(){
   Nodo aux = head;
   while( aux != null ){
         System.out.print( aux.getData() + "; " );
         aux = aux.getNext();
   }
   System.out.println();
    ...
}
```

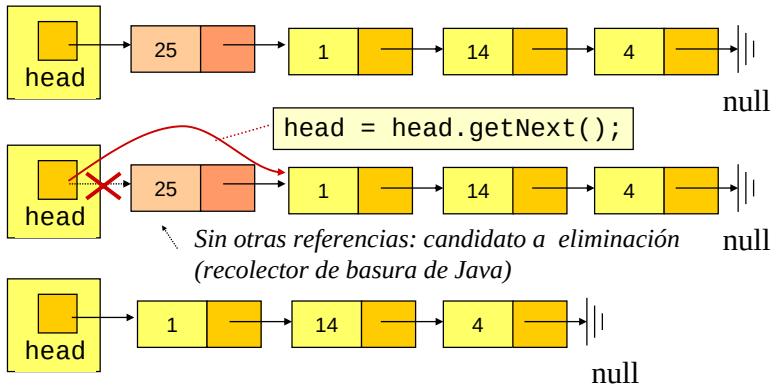

Búsqueda en la lista

- Retorna true si el elemento está contenido en la lista

```
public class Lista{
  ...
  public boolean estaContenido(int data){
      Nodo aux = head;
      while( aux != null ){
         if( data == aux.getData() )
            return true;
         aux = aux.getNext();
      }
      return false;
  }
  ...
}
```

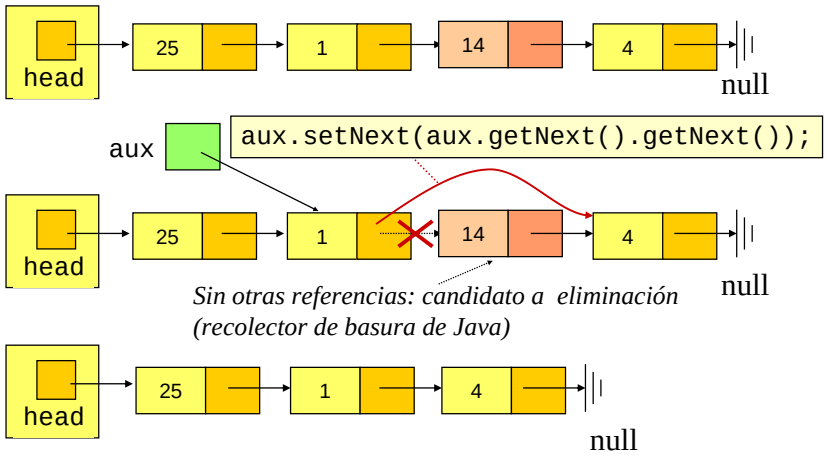

Eliminación de un elemento

- Requiere identificar el nodo a borrar.
- Caso 1: es el primer nodo de la lista



Eliminación de un elemento

- Caso 2 (general): no es el primer nodo de la lista



Eliminación de un elemento

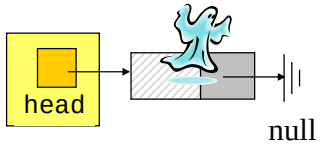
```
public class Lista{
  ...
  public boolean eliminar(int data){
     if( head != null)
        if( head.getData() == data ){
              head = head.getNext();
              return true;
        }else{
              Nodo aux = head;
              while( aux.getNext() != null ){
                 if( aux.getNext().getData() == data ){
                     aux.setNext( aux.getNext().getNext() );
                     return true;
                 }
                 aux = aux.getNext();
              }
        }
     return false;
  }
     ...
}
```


Simplificación del esquema propuesto: uso de un nodo "fantasma"

- En el esquema propuesto se deben hacer excepciones al insertar y eliminar el nodo del comienzo de la lista.
- El manejo se simplifica si se utiliza un nodo "fantasma":
  - Es un nodo siempre presente en la lista
  - Su contenido es irrelevante (el valor u objeto contenido no forma parte de la lista).

Lista simplemente enlazada con nodo fantasma

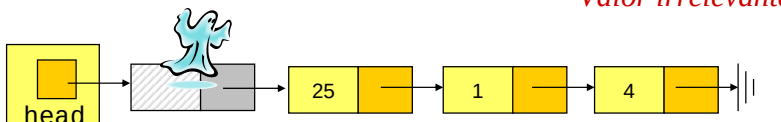
- Lista vacía:




```
public class Lista{
  Nodo head;
  public Lista(){
    head = new Nodo(0, null);
  }
  ...
}
```

Valor irrelevante

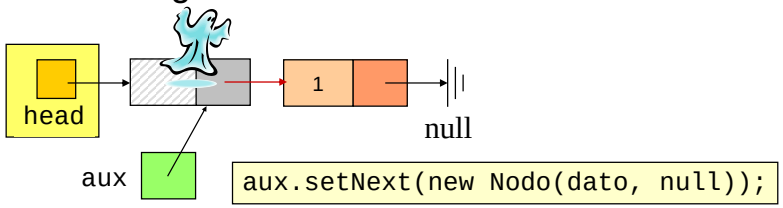
- Lista con elementos:



Primer elemento de la lista

Operación de inserción en la lista con nodo fantasma

- La inserción del primer elemento de la lista entra en el caso general:

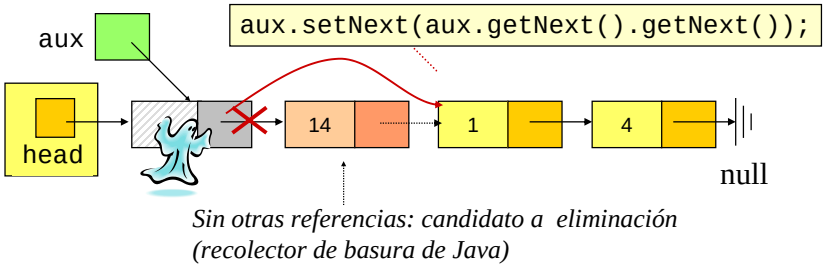


Clase Lista

```
public void agregarAlFinal(int dato){
      Nodo aux = head;
      while( aux.getNext() != null)
            aux = aux.getNext();
      aux.setNext( new Nodo(dato, null) );
      }
   }
```


Eliminación del primer elemento en la lista con nodo fantasma

- La eliminación del primer elemento de la lista entra en el caso general:



Sin otras referencias: candidato a eliminación (recolector de basura de Java)

Eliminación del primer elemento en la lista con nodo fantasma (cont.)

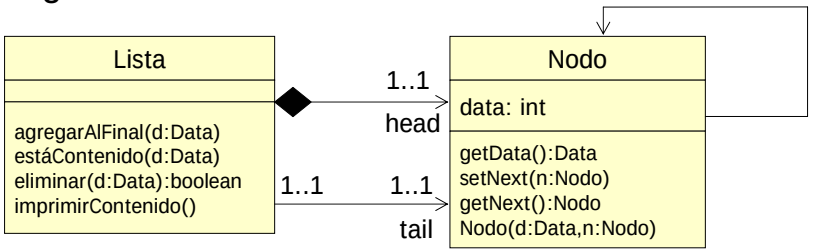
```
public boolean eliminar(int data){
        Nodo aux = head;
        while( aux.getNext() != null ){
           if( aux.getNext().getData() == data ){
               aux.setNext( aux.getNext().getNext() );
               return true;
           }
           aux = aux.getNext();
        }
        return false;
}
```


Mejora al procedimiento de inserción de elementos al final de la lista

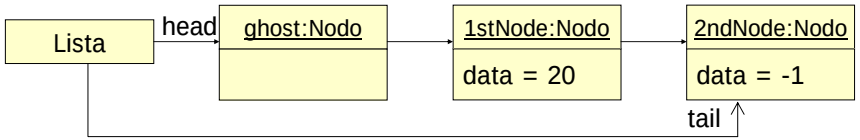
- El procedimiento descrito anteriormente requiere que todas las veces sea encontrado el último elemento de la lista.
- Más conveniente: tener una variable de instancia que siempre referencie al último elemento de la lista.
- Esto aplica a listas con o sin nodo fantasma (con pequeños cambios).

Mejora al procedimiento de inserción de elementos al final de la lista (cont.)

- Diagrama de clases:

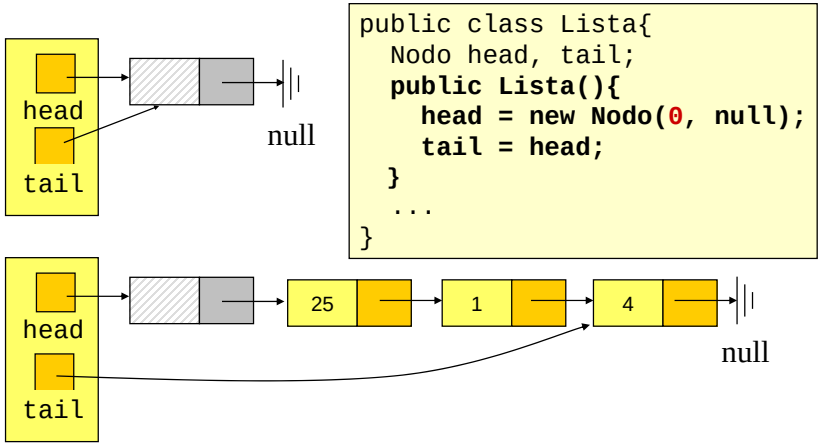


- Diagrama de objetos:



Mejora al procedimiento de inserción de elementos al final de la lista (cont.)

- La variable de instancia **`tail`** mantiene siempre la referencia al último elemento:



Mejora al procedimiento de inserción de elementos al final de la lista (cont.)

- El método **`agregarAlFinal`** ya no requiere recorrer la lista para ubicar el último nodo:

```
public void agregarAlFinal(int dato){
      Nodo aux = new Nodo(dato, null) ;
      tail.setNext( aux );
      tail = aux;
}
```

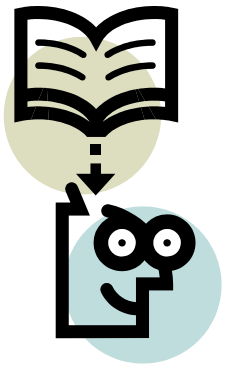
Versión con nodo fantasma

- La varaible **`tail`** es actualizada después de la inserción.
- Notar que el procedimiento de eliminación debe actualizar la referencia **`tail`** si se remueve el último nodo de la lista.

Inserción en orden/al inicio

- Ejercicio 1: implemente el método:
  `agregarEnOrden(int dato)`
  que recibe un entero y lo agrega en orden ascendente a la lista.

- Ejercicio 2: implemente el método
  `agregarAlInicio(int dato)`
  que recibe un entero y lo agrega como primer elemento.

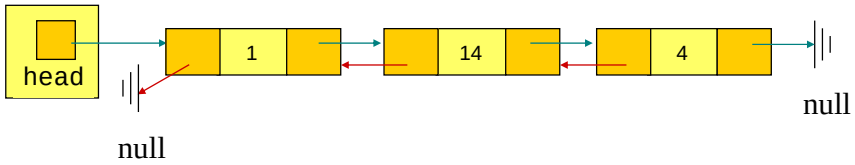


Resumen listas simplemente enlazadas

- Útiles para guardar un número no predefinido de elementos.
- Distintas disciplinas para mantener los datos ordenados (y para removerlos).
- El acceso a los nodos es secuencial; el recorrido es en una sola dirección (Ejercicio: confrontar con arreglos)

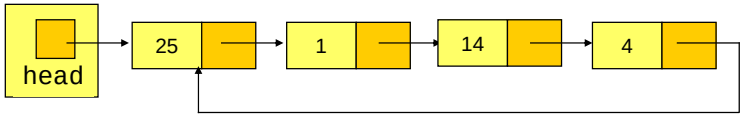
Listas doblemente enlazadas

- Están diseñadas para un acceso fácil al nodo siguiente y al anterior.
- Cada nodo contiene dos referencias: una apuntando al nodo siguiente, y otra apuntando al nodo anterior.
- El acceso a los nodos sigue siendo secuencial.
- La técnica del nodo fantasma puede ser útil también en este tipo de lista.



Comentarios finales sobre estructuras elementales

- Estar abierto a definir y utilizar otras estructuras.
- Ejemplo: Lista simplemente enlazada y circular



escuelaingenieríaindustrial
PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO

Colecciones implementadas sobre estructuras básicas

Franco Guidi Polanco
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso, Chile
fguidi@ucv.cl

Interfaces versus implementación

- En la sección anterior estudiamos estructuras elementales para implementar colecciones.
- En esta sección estudiaremos colecciones clásicas, desde dos perspectivas:
  - La interfaz de la colección (cómo se utiliza)
  - La implementación de la colección (cómo se construye)
- Una misma interfaz puede ser soportada por múltiples implementaciones
- La eficiencia en la operación de una colección va a depender de su implementación

Listas

- Representa una colección conformada por una secuencia finita y ordenada de datos denominados **elementos**.
- Ordenada implica que cada elemento tiene una **posición.**
- Los elementos corresponden a un **tipo de dato**.
- La lista está **vacía** cuando no contiene elementos.
- El número de elementos se denomina **largo** de la lista.
- El comienzo de la lista es llamado **cabeza** (head), y el final **cola** (tail).
- Notación: $(a_1, a_2, ..., a_n)$.

Lista: versión clásica

- Además de los datos, el estado de la lista contiene una identificación (referencia) al “dato actual”.

  (12, 22, 50, 30).
- La lista provee operaciones para:
  - Cambiar (modificar la referencia) del dato actual
  - Retornar el dato actual
  - Eliminar el dato actual
  - Modificar el dato actual
  - Insertar un dato en la posición del dato actual
  - Borrar toda la lista, buscar elementos en la lista, contar sus elementos

Interfaces para la versión clásica de la lista

- No hay una única definición formal de interfaz.
- Estudiaremos dos extremos:
  - Interfaz elemental
  - Interfaz extendida

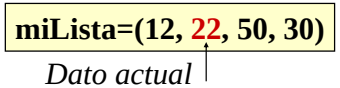
Interfaz elemental para la lista clásica

```
public interface List {
  public void clear();              // Elimina todos los elem.
  public void insert(Object item); // Inserta elem. en act.
  public Object remove();           // Saca/retorna elem.
  public void setFirst();           // Setea act. en 1ra pos.
  public void next();               // Mueve act. a sig. pos.
  public int length();              // Retorna largo
  public void setValue(Object val);// Setea elemento
  public Object currValue();        // Retorna elemento
  public boolean isEmpty();         // True: lista vacía
  public boolean eol();             // True: act en end of list
  public String toString();         // Retorna lista de elem.
}
```

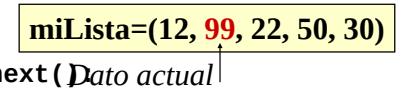
Nota: pos.=posición; act.=posición actual; sig.:siguiente; prev.:previa

Ejemplo de uso de una lista (versión clásica)

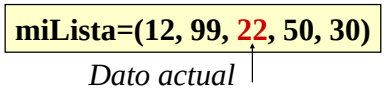
- sea la siguiente lista:



- Operaciones:
  - `miLista.insert(99)` :

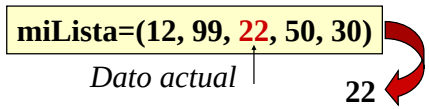


  - `miLista.next()`:

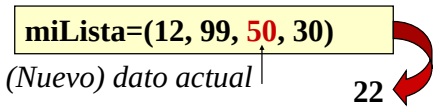


Ejemplo de uso de una lista (versión clásica)

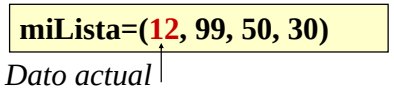
- `miLista.currValue()`:



- `miLista.remove()`:



- `miLista.setFirst()`:



Interfaz extendida para la lista clásica

```
public interface ExtendedList {
  public void clear();              // Elimina todos los elem.
  public void insert(Object item); // Inserta elem. en act.
  public void append(Object item); // Agrega elem. al final
  public Object remove();           // Saca/retorna elem.
  public void setFirst();           // Setea act. en 1ra pos.
  public void next();               // Mueve act. a sig. pos.
  public void prev();               // Mueve act. a pos. prev.
  public int length();              // Retorna largo
  public void setPos(int pos);      // Setea act. a pos
  public void setValue(Object val);// Setea elemento
  public Object currValue();        // Retorna elemento
  public boolean isEmpty();         // True: lista vacía
  public boolean eol();             // True: act en end of list
  public String toString();         // Retorna lista de elem.
}
```

Nota: pos.=posición; act.=posición actual; sig.:siguiente; prev.:previa

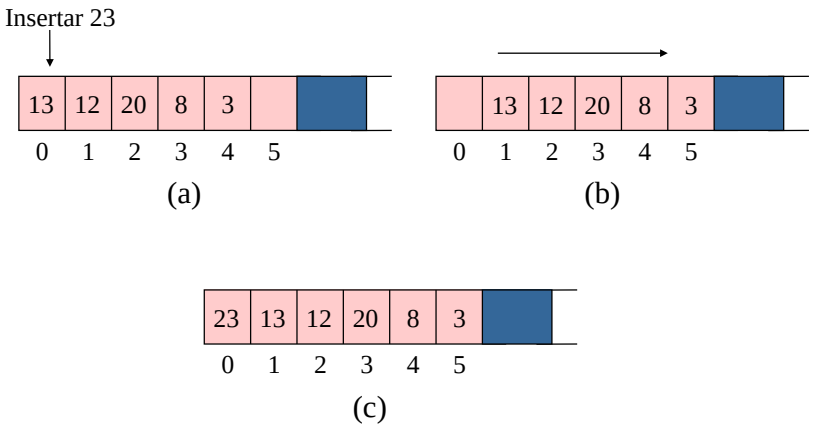
Implementación de la lista clásica

- La conveniencia de una u otra implementación depende de las operaciones definidas en la interfaz.
- Interfaz elemental:
  - Implementación basada en lista simplemente enlazada
  - Implementación basada en arreglos
  - Implementación basada en lista doblemente enlazada (sobredimensionada)
- Interfaz extendida:
  - Implementación basada en arreglos
  - Implementación basada en lista doblemente enlazada
  - Implementación basada en lista simplemente enlazada

Implementación de la lista clásica basada en arreglos

- Usa un arreglo para almacenar los elementos de la lista.
- Los elementos son almacenados en posiciones contiguas en el arreglo.
- El elemento “i” de la lista se almacena en la celda “i-1” del arreglo.
- La cabeza de la lista siempre está en la primera posición del arreglo (0).
- El máximo número de elementos en la lista se define al crear el arreglo.

Inserción en lista basada en arreglos



Comparación entre implementaciones de listas

- Las listas basadas en arreglos tienen la desventaja de que su número de elementos debe ser predeterminado.
- Cuando estas listas tienen pocos elementos, se desperdicia espacio.
- Las listas enlazadas no tienen límite de número máximo de elementos (mientras la memoria lo permita).

Comparación entre implementaciones de listas

- Las listas basadas en arreglos son más rápidas que aquellas basadas en listas enlazadas para el acceso aleatorio por posición.
- Las operaciones de inserción y eliminación son más rápidas en las listas enlazadas.
- En general, si el número de elementos que contendrá una lista es muy variable o desconocido, es mejor usar listas enlazadas.

Uso de listas

- Supongamos las siguientes implementaciones de la lista:
  - `LList`, implementa `List` mediante una lista simplemente enlazada
  - `AList`, implementa `List` mediante un arreglo
- Referenciamos la lista por medio de su interfaz:

```
...
List lista = new LList(); // Implementación seleccionada
lista.insert( “Hola” );
lista.insert( “chao” );
lista.setFirst();
while( !lista.eol() ){
       System.out.println( (Sring)lista.currValue());
       lista.next();
}
lista.setFirst();
String eliminado = (String) lista.remove()
...
```


Uso de listas (cont.)

- Si en el programa anterior ahora se desea utilizar otra implementación de lista, sólo debe cambiarse la clase a instanciar.

```
...
List lista = new AList(); // Implementación seleccionada
lista.insert( “Hola” );
lista.insert( “chao” );
lista.setFirst();
while( !lista.eol() ){
       System.out.println( (Sring)lista.currValue());
       lista.next();
}
lista.setFirst();
String eliminado = (String) lista.remove()
...
```


Pilas

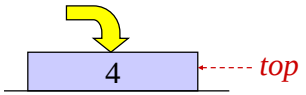
- Es un tipo restringido de lista, en donde los elementos sólo pueden ser insertados o removidos desde un extremo, llamado **top**.
- Se le llama también **Stack** o **Lista LIFO** (Last In, First Out).
- La operación para insertar un nuevo elemento se denomina **push**.
- La operación para remover un elemento se denomina **pop**.

Interfaz para la Pila

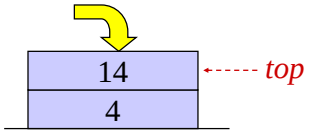
```
public interface Stack {
  public void clear();     // Remueve todos los objetos
  public void push(Object it); // Agrega objeto al tope
  public Object pop();         // Saca objeto del tope
  public Object topValue();// Retorna objeto en el tope
  public boolean isEmpty();    // True: pila vacía
}
```


Ejemplo de uso de una pila

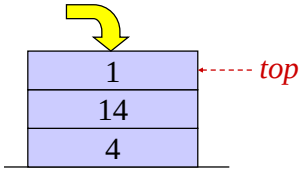
- **pila.push(4):**



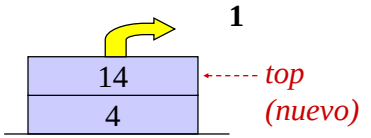
- **pila.push(14):**



- **pila.push(1):**

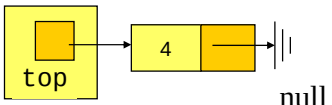


- **pila.pop():**

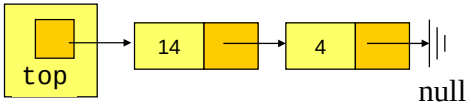


Implementación de una pila mediante lista simplemente enlazada

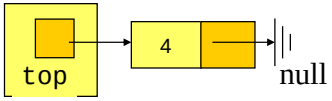
pila.push(4)



pila.push(14)

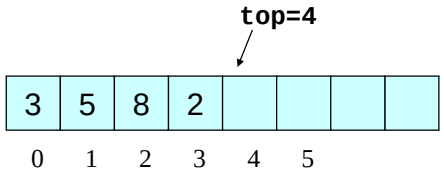


pila.pop()



Se inserta y remueve siempre el primer elemento:
no se requiere nodo fantasma

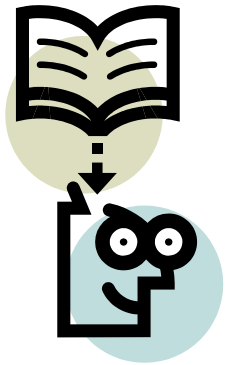
Implementación de una pila mediante un arreglo



- En esta clase, por conveniencia, se usa una variable **top**, que siempre está 1 posición más adelante del elemento “superior” de la pila.

Ejercicio pilas

- Proponga implementaciones de la pila:
  - Basada en un arreglo
  - Basada en una lista simplemente enlazada



Colas

- Es un tipo restringido de lista, en donde los elementos sólo pueden ser agregados al final, y removidos por el frente.
- Se le llama también **Queue** o **Lista FIFO** (First In, First Out).
- La operación para agregar un nuevo elemento se denomina **enqueue**.
- La operación para remover un elemento se denomina **dequeue**.

Interfaz para la Cola

```
public interface Queue {
  public void clear(); // Remueve todos los objetos
  public void enqueue(Object it);// Agrega obj. al final
  public Object dequeue(); // Saca objeto del frente
  public Object firstValue(); // Retorna obj. del frente
  public boolean isEmpty(); // Retorna V si cola vacía
}
```


Ejemplo de uso de una cola

- `cola.enqueue( 20 )`

  cola=(20)
- `cola.enqueue( 15 )`

  cola=(20, 15)
- `cola.enqueue( 40 )`

  cola=(20, 15, 40)
- `cola.dequeue()`

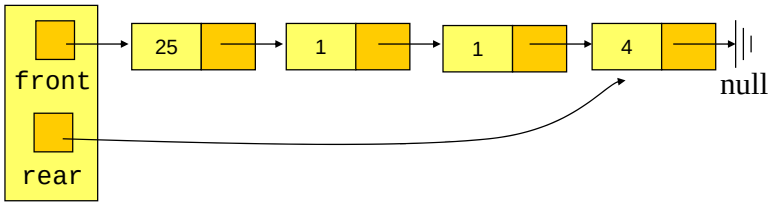
  cola=(15, 40)

  20
- `cola.dequeue()`

  cola=(40)

  15

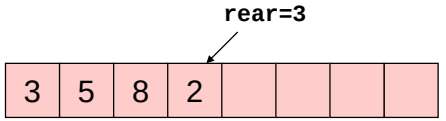
Cola basada en lista simplemente enlazada



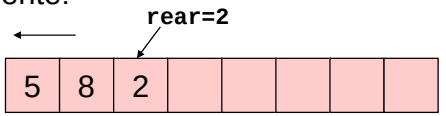
- Es conveniente mantener una referencia al último elemento de la cola (facilita operación **enqueue**).

Cola basada en arreglos (cont.)

- Aproximación simple: almacenar los "n" elementos de la cola en las "n" primeras posiciones del arreglo.



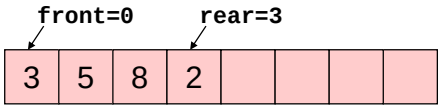
Sacar del frente:



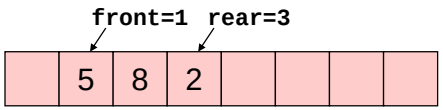
Problema: lentitud de procedimiento dequeue (sacar primer elemento).

Cola basada en arreglos (cont.)

- Aproximación mejorada: al hacer dequeue, no desplazar elementos, sino asumir que el frente de la cola se desplaza.



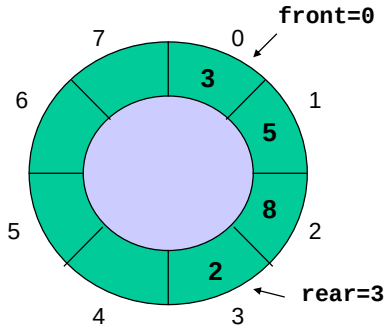
Sacar del frente:



Problema: Al sacar y agregar elementos, **rear** llega a la última posición del arreglo y la cola no puede crecer, aun cuando existan posiciones libres al comienzo.

Cola basada en arreglos* (cont.)

- Arreglo circular: Pretender que el arreglo es "circular", y permitir que la cola continúe directamente de la última posición del arreglo a la primera.



Función de avance

Dados:
- **pos**: posición actual
- **size**: tamaño arreglo

```
pos=(pos+1);
if (pos>=size)
   pos=0;
```

Cola basada en arreglos* (cont.)

- Problema de arreglo circular:
  - Si **`front`** es igual a **`rear`**, implica que hay 1 elemento.
  - Luego, **`rear`** está una posición detrás de **`front`** implica que la cola está vacía.
  - Pero si la cola está llena, **`rear`** también está una posición detrás de **`front`**.
- Problema: ¿Cómo reconocer cuando una cola está vacía o llena?
- Solución: Usar un arreglo de n posiciones, para almacenar como máximo n-1 elementos.

Cola basada en arreglos* (cont.)

- Por conveniencia, se usa una variable **`front`** para apuntar a la posición precedente al elemento frontal.
- La cola está vacía cuando **`front=rear`**.
- La cola está llena cuando **`rear`** está justo detrás de **`front`**, o sea cuando la función de avance indica que de aumentar **`rear`** en 1, se llegaría a **`front.`**